

way of the hax0r
n00b to console punk

Chapter 8

Who Can Do What?

Sooner or later a command answers “Permission denied”. The file is right there in the listing, the spelling is right, and the shell still refuses. The refusal is not a bug. The operating system checks who you are and what that identity is allowed to do with each file, and the check applies to every program you run, including the shell.

The concepts in this chapter are:

- **users** and **groups**, the identities the operating system tracks;
- **ownership**: every file has an owner, and on UNIX a group;
- UNIX **permissions**: read, write, and execute for owner, group, and others;
- Windows **access control lists** (ACLs), a more detailed model;
- what makes a file **executable** on each system;
- **elevation**: running with administrator rights, and why to avoid it by default.

The examples inspect permissions and change them on files you own inside the practice directory. They do not ask you to run anything as an administrator.

Users and Groups

The operating system identifies you by a **user account**. Every process you start, including the shell, runs as that user, and every file you create is owned by it. A **group** is a named set of users, so that access can be granted to a team without listing each member on each file.

Find out who you are:

Bash/zsh:

```
$ whoami  
$ id
```

`id` prints your numeric user ID, your primary group, and every group you belong to. The `sudo` or `wheel` group, where present, is the one that allows elevation, which the last section covers. [1]

PowerShell and Command Prompt:

```
$ whoami
$ whoami /groups
```

On Windows, `whoami` reports `computer\user` or `domain\user`, and `/groups` lists group memberships, including whether the session is a member of Administrators. [2] PowerShell on Linux and macOS runs that system's `whoami`.

Two users are special. On UNIX-style systems, **root** can do anything, and files that belong to the system are owned by it. On Windows, the **Administrators** group plays that role, and the built-in **SYSTEM** account owns much of the operating system. Your everyday account should be neither.

Ownership and UNIX Permissions

On Linux and macOS, every file and directory has an owner, a group, and nine permission bits. The long listing shows all of it:

```
$ ls -l notes.txt
-rw-r--r-- 1 student student 33 Sep 20 10:00 notes.txt
```

Read the first field in four parts:

Characters	Meaning
-	Type: - file, d directory, l symbolic link
rw-	The owner can read and write, not execute
r--	Members of the group can read
r--	Everyone else, others , can read

Then come the owner and group names. Your own listing shows your username and your primary group. [3]

The three permissions mean different things for files and directories:

Permission	On a file	On a directory
r read	View the contents	List the names inside
w write	Change the contents	Create, rename, and delete entries inside
x execute	Run it as a program	Enter it and reach the entries inside

Two consequences are easy to miss. Deleting a file needs write permission on its **directory**, not on the file. And a directory without x blocks access to everything inside, even if the entries themselves are readable.

Change Permissions with `chmod`

`chmod` changes the permission bits. It has a symbolic form and a numeric form, and both are common.

```
$ printf '#!/bin/sh\nnecho ran\n' > hello.sh
$ ls -l hello.sh
-rw-r--r-- 1 student student 20 Sep 20 10:05 hello.sh
$ chmod u+x hello.sh
$ ls -l hello.sh
-rwxr--r-- 1 student student 20 Sep 20 10:05 hello.sh
```

The symbolic form names who (u owner, g group, o others, a all), an operator (+ add, - remove, = set exactly), and the permissions. `chmod go-r hello.sh` removes read from group and others; `chmod a=r notes.txt` makes a file read-only for everyone. [4]

The numeric form gives three digits, one each for owner, group, and others, where read is 4, write is 2, and execute is 1, added together:

Digits	Permissions	Typical use
644	<code>rw-r--r--</code>	An ordinary file
755	<code>rwxr-xr-x</code>	A program or a directory
600	<code>rw-----</code>	A private file, such as an SSH key
700	<code>rwx-----</code>	A private directory

`chmod 755 hello.sh` and `chmod u=rwx,go=rx hello.sh` are the same request. The numeric form is what you will see in most instructions. [4]

New files do not start with every permission set. The **umask** is a per-session setting that removes permissions from new files; a common value of 022 produces 644 files and 755 directories. Run `umask` to see yours. [5]

Change Ownership with `chown`

`chown` changes the owner, and `chgrp` or `chown user:group` changes the group. Only root can give a file to another user, so most `chown` commands you see are run with `sudo`:

```
$ chown root notes.txt
chown: changing ownership of 'notes.txt': Operation not permitted
```

You do not need `chown` in this chapter. Recognize it in instructions, and ask why a file's owner needs to change before running it. [6]

Windows Access Control Lists

Windows does not use nine bits. Each file and directory carries an **access control list**: a list of entries, each naming a user or group, a set of rights such as read, write, modify, or full control, and whether the entry allows or denies them. Entries are usually **inherited** from the parent directory, which is how a whole tree gets consistent permissions. [7]

`icacls` shows and edits ACLs from PowerShell or Command Prompt:

```
$ icacls notes.txt
notes.txt BUILTIN\Administrators:(I)(F)
          NT AUTHORITY\SYSTEM:(I)(F)
          LAPTOP\student:(I)(F)
```

(I) marks an inherited entry and (F) means full control; (R), (W), and (M) are read, write, and modify. `icacls notes.txt /grant "Users:(R)"` adds an entry, and `/deny` adds a deny entry, which wins over any allow. [8]

PowerShell on Windows also has `Get-Acl` and `Set-Acl`, which return the ACL as an object you can inspect with `Get-Member`:

```
$ Get-Acl notes.txt | Format-List Owner, AccessToString
```

`Get-Acl` and `Set-Acl` exist only on Windows; PowerShell on Linux and macOS does not have them, and the `Mode` column in its listings shows UNIX permissions instead. [9]

The graphical route is the **Security** tab in a file's Properties window, which shows the same list. Use whichever is easier to read, but expect instructions to use `icacls`.



Wut: "Permission denied" on Windows can come from an ACL, from a file being open in another program, or from a file marked as downloaded from the internet. Check `icacls` first, then close programs that might hold the file, then look at the Unblock option in the file's Properties.

What Makes a File Executable

The two families answer this question differently, and the difference explains several confusing errors.

On UNIX-style systems, a file runs as a program only if its execute bit is set. For a script, the first line, the **shebang**, names the interpreter: `#!/bin/sh`, `#!/usr/bin/env python3`, or `#!/usr/bin/env pwsh`. The kernel reads that line and starts the interpreter with the script as its argument. [10]

```
$ ./hello.sh
bash: ./hello.sh: Permission denied
$ chmod +x hello.sh
$ ./hello.sh
ran
$ bash hello.sh
ran
```

The `./` is required because the current directory is not on `PATH`, which Chapter 2 explained. The last line shows the other way to run a script without the execute bit: give it to the interpreter directly. [11]

On Windows, the file **extension** decides. `.exe`, `.bat`, `.cmd`, and the others listed in the `PATHEXT` environment variable run as programs; a file without one of those extensions opens in whatever program is associated with it. [12]

PowerShell scripts, .ps1 files, add one more rule: the **execution policy**. The default on Windows client editions is `Restricted`, which runs no scripts at all, and many organizations set `RemoteSigned`, which runs local scripts and only signed downloaded ones. `Get-ExecutionPolicy -List` shows the setting at each scope, and `Set-ExecutionPolicy -Scope CurrentUser RemoteSigned` changes it for your account without administrator rights. On Linux and macOS the policy is `Unrestricted` and not enforced. [13]

Windows also marks files downloaded by a browser with a **zone identifier**, and PowerShell treats such scripts as remote. `Unblock-File script.ps1` removes the mark after you have read the script. [14]

Elevation

Some operations need more than your account has: installing software system-wide, editing files under `/etc` or `C:\Windows`, and changing another user's files. **Elevation** runs one command, or one session, with those extra rights.

Bash/zsh:

```
$ sudo apt update
[sudo] password for student:
```

`sudo` runs a single command as root after checking that your account is allowed to, usually through membership of the `sudo` or `wheel` group. It asks for your own password, not root's, and remembers the answer for a few minutes. `sudo -i` opens a whole root shell; leave it as soon as the task is done. [15]

PowerShell and Command Prompt:

Windows uses **User Account Control** (UAC). Even an account in the Administrators group runs with ordinary rights until it asks to elevate, which produces the consent prompt. To elevate a shell, right-click Windows Terminal or PowerShell and choose **Run as administrator**, or start it from an existing session:

```
$ Start-Process pwsh -Verb RunAs
```

The elevated window is a separate session; the one you started it from is unchanged. Windows 11 also ships a `sudo` command, off by default and enabled under **Settings > System > For developers**, which runs one command elevated. [16], [17]

Use elevation for the specific command that needs it, and no more. An elevated shell makes every typo as powerful as the intended command, and files created while elevated are owned by root or Administrators, which causes the next round of "Permission denied" in your ordinary session.



Trap: Adding `sudo` to a command because it failed is not troubleshooting. First read the error and check ownership and permissions with `ls -l` or `icacls`. If the file is yours, fix the permission bits; if it belongs to the system, ask whether you should be changing it at all.

Key Points

- Every process runs as a user, every file has an owner, and the operating system checks one against the other.
- UNIX permissions are read, write, and execute for owner, group, and others; `ls -l` shows them and `chmod` changes them, symbolically or with digits such as 644 and 755.
- Deleting a file needs write permission on the directory; entering a directory needs execute permission on it.
- Windows uses access control lists with inherited allow and deny entries; `icacls` shows and edits them, and `Get-Acl` exists only on Windows.
- UNIX runs a file if its execute bit is set and its shebang names an interpreter; Windows decides by extension, and PowerShell adds an execution policy for scripts.
- `sudo` and Run as administrator elevate one command or session; use them for the specific step that needs them.

Level Up

n00bs finish the chapter here. You can read a permission listing, change the bits on your own files, and explain why a command was denied. Hackers investigate the model behind the bits.

Read about the **setuid** bit and find a setuid program on your system with `find / -perm -4000 -type f 2>/dev/null`. Explain why `passwd` needs it and why setuid scripts are refused.

Explore **ACLs on Linux** with `getfacl` and `setfacl`, and compare them with the Windows model. Then look at **file attributes** with `lsattr` and `chattr`, and at macOS **flags** with `ls -lO` and `chflags`.

Learn how `sudo` decides who may run what by reading `/etc/sudoers` documentation, and why editing it with `visudo` matters. On Windows, read about **integrity levels** and explain what `whoami /groups` shows under “Mandatory Label”.

Investigate how containers and sandboxes such as Docker, Flatpak, and Windows Sandbox change the answer to “who can do what”, and what user a process inside one appears to be.

Exercises

1. **What does this do?** Predict the listing after `chmod 640 notes.txt`, then run it and check.
2. **What does this do?** Predict what `chmod -x .` does to your ability to list, enter, and create files in the practice directory, then test it and restore with `chmod +x ..`.
3. **What is the problem?** A script downloaded on Windows fails with “running scripts is disabled on this system”. Name the setting responsible and give the command that changes it for the current user only.
4. **What is the problem?** After running an editor with `sudo` to fix a file, the user can no longer save that file from their normal editor. Explain what changed and how to fix it.
5. **Why is it like this?** A user can delete a file they cannot read. Explain which permission on which object allows that.
6. **Why is it like this?** `./hello.sh` says “Permission denied” but `bash hello.sh` runs. Explain what each form asks the operating system to do.
7. **How do you** make a directory readable by everyone but writable only by you, using both `chmod` forms?

8. **How do you** find out whether your current PowerShell session on Windows is elevated, without trying something that needs elevation?

References

- [1] GNU Project, “GNU coreutils manual: Id: Print user identity.” Available: https://www.gnu.org/software/coreutils/manual/html_node/id-invocation.html
- [2] Microsoft, “Windows commands: whoami.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/whoami>
- [3] GNU Project, “GNU coreutils manual: File permissions.” Available: https://www.gnu.org/software/coreutils/manual/html_node/File-permissions.html
- [4] GNU Project, “GNU coreutils manual: Chmod: Change access permissions.” Available: https://www.gnu.org/software/coreutils/manual/html_node/chmod-invocation.html
- [5] GNU Project, “Bash reference manual: Bourne shell builtins (umask).” Available: https://www.gnu.org/software/bash/manual/html_node/Bourne-Shell-Builtins.html
- [6] GNU Project, “GNU coreutils manual: Chown: Change file owner and group.” Available: https://www.gnu.org/software/coreutils/manual/html_node/chown-invocation.html
- [7] Microsoft, “Access control lists (windows authorization).” Available: <https://learn.microsoft.com/en-us/windows/win32/secauthz/access-control-lists>
- [8] Microsoft, “Windows commands: icacls.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/icacls>
- [9] Microsoft, “Get-acl.” Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.security/get-acl>
- [10] GNU Project, “Bash reference manual: Shell scripts.” Available: https://www.gnu.org/software/bash/manual/html_node/Shell-Scripts.html
- [11] GNU Project, “Bash reference manual: Command search and execution.” Available: https://www.gnu.org/software/bash/manual/html_node/Command-Search-and-Execution.html
- [12] Microsoft, “About_environment_variables.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_environment_variables
- [13] Microsoft, “About_execution_policies.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies
- [14] Microsoft, “Unblock-file.” Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/unblock-file>
- [15] Todd C. Miller, “Sudo(8) manual page.” Available: <https://www.sudo.ws/docs/man/sudo.man/>
- [16] Microsoft, “How user account control works.” Available: <https://learn.microsoft.com/en-us/windows/security/application-security/application-control/user-account-control/how-it-works>

- [17] Microsoft, “Sudo for windows.” Available: <https://learn.microsoft.com/en-us/windows/sudo/>